Sort in Java

0.2 术语说明

- **稳定**：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
- **不稳定**：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；
- **内排序**：所有排序操作都在内存中完成；
- **外排序**：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；
- **时间复杂度**：一个算法执行所耗费的时间。
- **空间复杂度**：运行完一个程序所需内存的大小。

0.3 算法总结

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

图片名词解释：

- n: 数据规模
- k: “桶”的个数
- In-place: 占用常数内存，不占用额外内存
- Out-place: 占用额外内存

1.Selection sort
2.Insertion sort
3.Heap sort
4.Merge sort
5.Quick sort
6.Bubble sort
7.Radix sort
8.Shell sort
9.Bucket sort

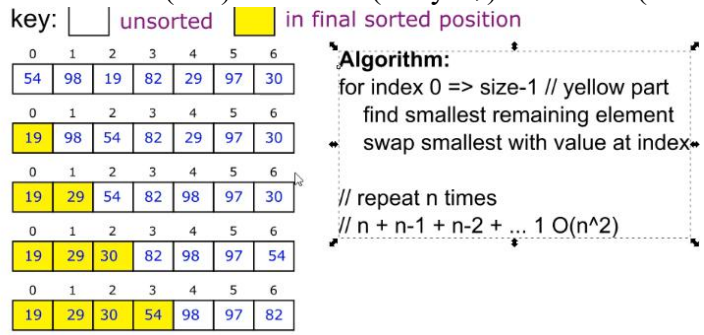
Out-place: Merge sort, Counting sort, Bucket sort, Radix sort
Comparison sorting algorithm 是 array of elements 能相互比较，element 是 string 也可以

比较排序有: Bubble sort, Heap sort, Merge sort, Quick sort,Insertion sort, Shell sort, Selection Sort 适用一切排序
非比较排序有: Radix sort, bucket sort, counting sort，相对时间复杂度低，但要 extra memory

Stable:如果 a 原本在 b 前面，而 a=b，排序之后 a 仍然在 b 前面
Unstable：如果 a 原本在 b 前面，而 a=b，排序之后 a 可能会出现 b 的后面

一. Selection sort

worst&best O(n^2)　unstable(array 时)，　stable(linked list 时)



54 98 19 82 29 97 30
把第一个数(54)作为 sorted position，再往后 search array 里最小的数和 54 换位置，再把 98 作为放到 sorted position，再重复搜索整个 array 里最小的数字和 98 换

```
public static void selectionSort(int[] array) {
for (int i = 0; i < array.length; i++) {
int min = array[i];
int minId = i;
for (int j = i+1; j < array.length; j++) {
  if (array[j] < min) {
  min = array[j];
  minId = j;
}
}
// swapping
int temp = array[i];
array[i] = min;
  array[minId] = temp; } }
```

Recursive

```java
// Recursive selection sort. n is size of a[] and index
// is index of starting element.
static void recurSelectionSort(int a[], int n, int index)
{

    // Return when starting and size are same
    if (index == n)
       return;

    // calling minimum index function for minimum index
    int k = minIndex(a, index, n-1);

    // Swapping when index nd minimum index are not same
    if (k != index){
       // swap
       int temp = a[k];
       a[k] = a[index];
       a[index] = temp;
    }
    // Recursively calling selection sort function
    recurSelectionSort(a, n, index + 1);
}
```

Problem 1:

Update the following array contents to reflect 3 iterations of Selection Sort's outer loop:

{ 4, 9, 2, 3, 8, 5 }

{ 4, 9, 2, 3, 8, 5 }
{ 2, 9, 4, 3, 8, 5 }
{ 2, 3, 4, 9, 8, 5 }

二. Insertion sort

worst O(n^2)--每次都要把新元素放 sorted position 最前面　best O(n)　stable

sorted position 不是 final position
每次 sorted position 多包括一个 index，然后在 sorted position 里 sort--把新的 index 元素放在合适的位置

key: unsorted within sorted portion of array (not necessarily final positic

0	1	2	3	4	5	6
54	98	19	82	29	97	30
54	98	19	82	29	97	30
19	54	98	82	29	97	30
19	54	82	98	29	97	30
19	29	54	82	98	97	30

Algorithm:
for index 1 => size-1 // yellow part
　bubbleLeft(index)
　// repeately swaps index with
　// index-1 until either:
　//　index == 0 or element at index
　//　is larger than element at index-1

// O(n^2)

Problem 2

Update the followin
contents to reflect
of Insertion Sort's
{ 4, 9, 2, 3, 8

```
public static void insertionSort(int[] array) {
for (int i = 1; i < array.length; i++) {
int current = array[i];
  int j = i - 1;
while(j >= 0 && current < array[j]) {
array[j+1] = array[j];
j--; }
// at this point we've exited, so j is either -1
// or it's at the first element where current >= a[j]
array[j+1] = current;
} }
```

Recursive

```java
// Recursive function to sort an array using
// insertion sort
void insertionSortRecursive(int arr[], int n)
{
    // Base case
    if (n <= 1)
        return;

    // Sort first n-1 elements
    insertionSortRecursive( arr, n-1 );

    // Insert last element at its correct position
    // in sorted array.
    int last = arr[n-1];
    int j = n-2;

    /* Move elements of arr[0..i-1], that are
      greater than key, to one position ahead
      of their current position */
    while (j >= 0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}
```

Problem 2:

Update the following array contents to reflect 3 iterations of Insertion Sort's outer loop:

{ 4, 9, 2, 3, 8, 5 }

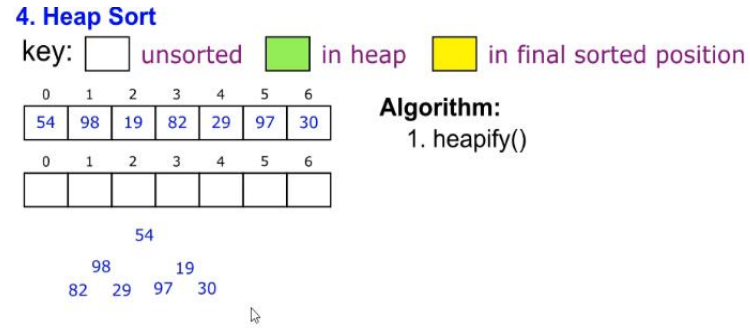
{ 2, 3, 4, 9, 8, 5 }

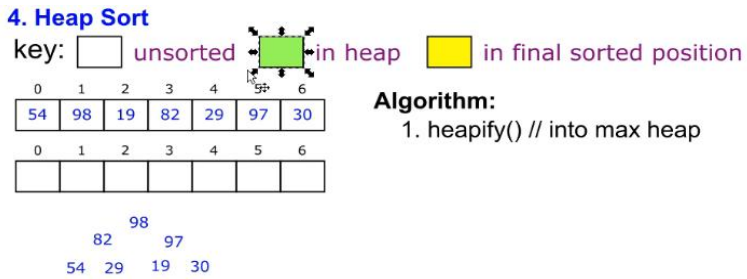
三. Heap sort

worst&best O(n*log2n) unstable

简单的说就是把 heap 先 heapify 成 max-heap，再把 root(最大 value)放到末尾处(sorted part)--和末尾处的 element 互换位置，（之后的操作不用考虑 sorted part）对 heap 进行 percolate down，这样 root 又是最大 value，重复之前步骤

1.先转换成 max-heap 结构，再 percolate down，再转换回 array 结构变成
98 82 97 54 29 19 30



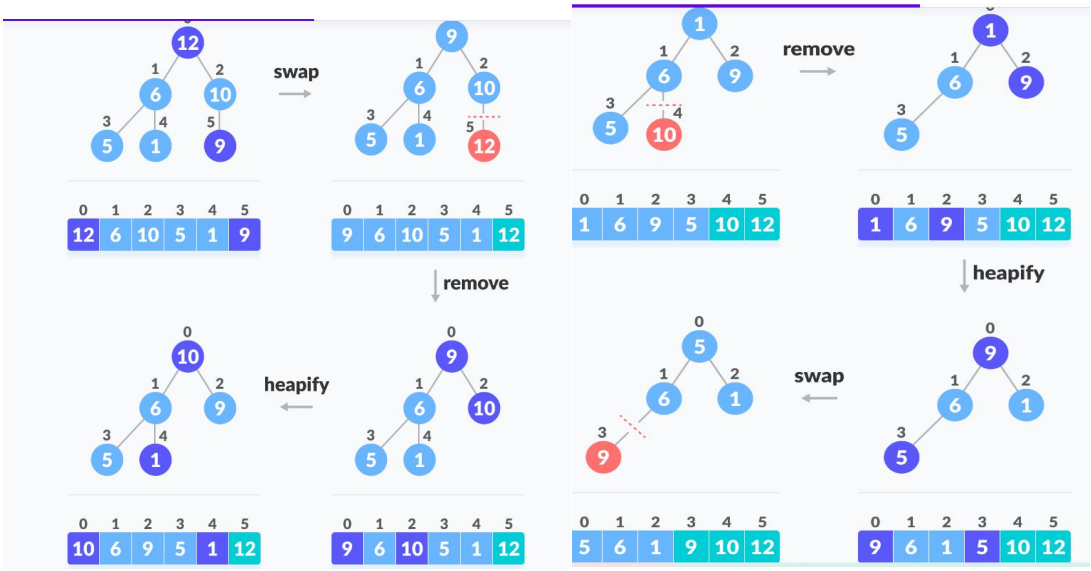


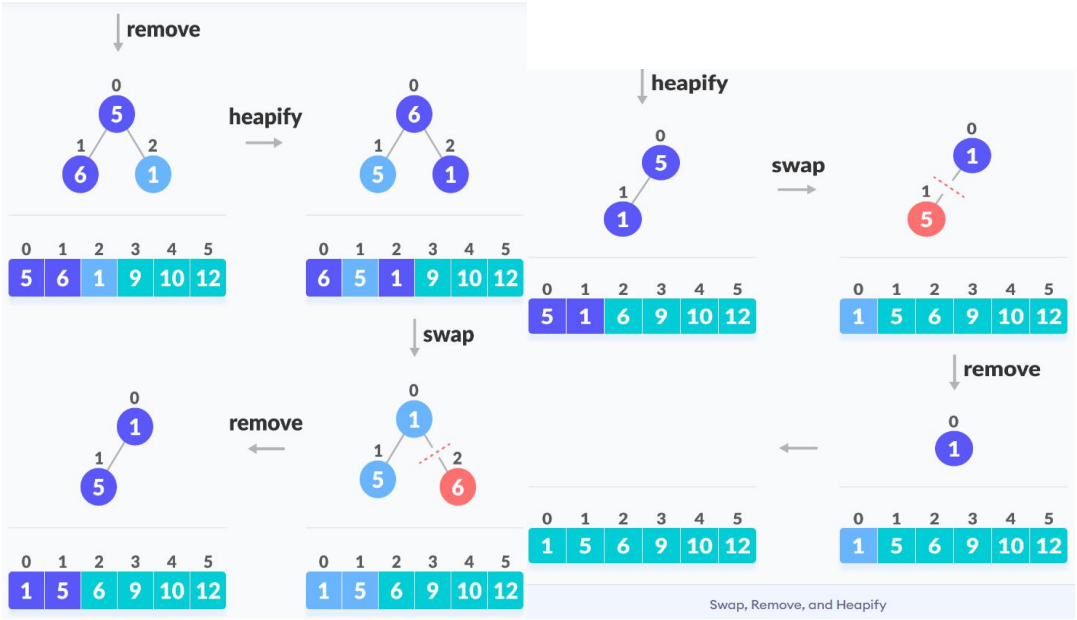
2. 开始 heap sort，把最大的 value 放最后

图从左到右

12

34


swap
remove
heapify
swap
remove
heapify


remove
heapify
swap
heapify
swap
remove
Swap, Remove, and Heapify

```java
static void heapify(int[] array, int length, int i) {
    int leftChild = 2*i+1;
    int rightChild = 2*i+2;
    int largest = i;

    // if the left child is larger than parent
    if (leftChild < length && array[leftChild] > array[largest]) {
        largest = leftChild;
    }

    // if the right child is larger than parent
    if (rightChild < length && array[rightChild] > array[largest]) {
        largest = rightChild;
    }

    // if a swap needs to occur
    if (largest != i) {
        int temp = array[i];
        array[i] = array[largest];
        array[largest] = temp;
        heapify(array, length, largest);
    }
}

public static void heapSort(int[] array) {
    if (array.length == 0) return;

    // Building the heap
    int length = array.length;
    // we're going from the first non-leaf to the root
    for (int i = length / 2-1; i >= 0; i--)
        heapify(array, length, i);

    for (int i = length-1; i >= 0; i--) {
        int temp = array[0];
        array[0] = array[i];
        array[i] = temp;

        heapify(array, i, 0);
    }
}
```

Problem 3:

max heap
ze-2
= heapRemove()

Update the following array contents to reflect 3 iterations of Heapify's outer loop (using percolate down):

{ 4, 9, 2, 3, 8, 5 }

ize-2
= heapRemove()

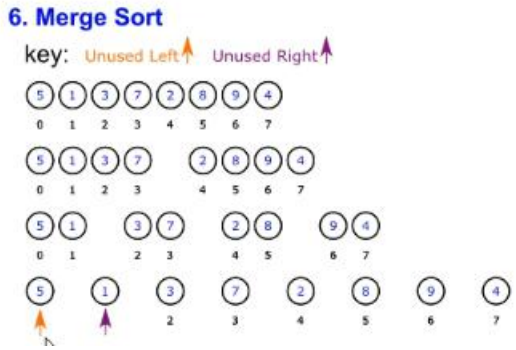
Update the following array contents to reflect 3 iterations of Heapify's outer loop (using percolate down):

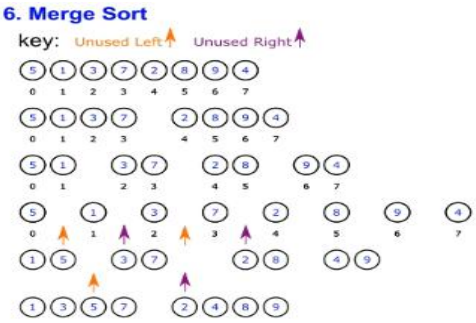
{ 4, 9, 2, 3, 8, 5 }

9 8 5 3 4 2 三次 swap

四．Merge sort

worst/best O(n*log2n)--outer recursive call O(log2n) inner loop--O(n),
如果是两个 sorted array 则是 O(n)
stable
把 array 二分直到每个子序只有一个 element，再合并，每次合并都伴随 sort



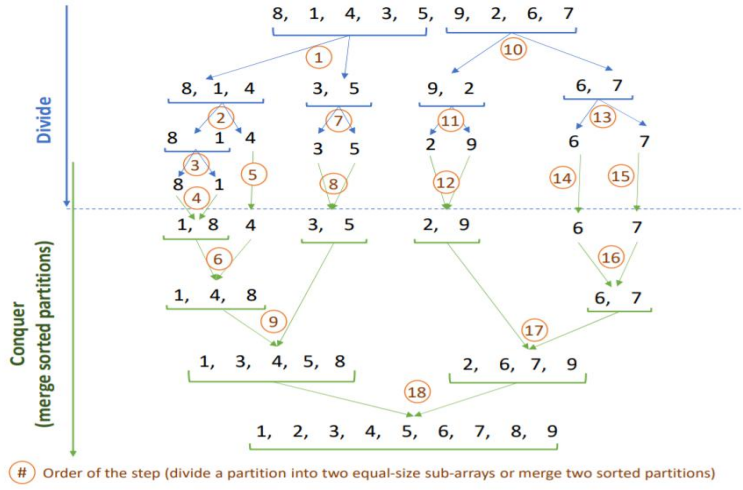


先合并 element 第 1 和 2 ，3 和 4，再合并 12 和 34，右边同理

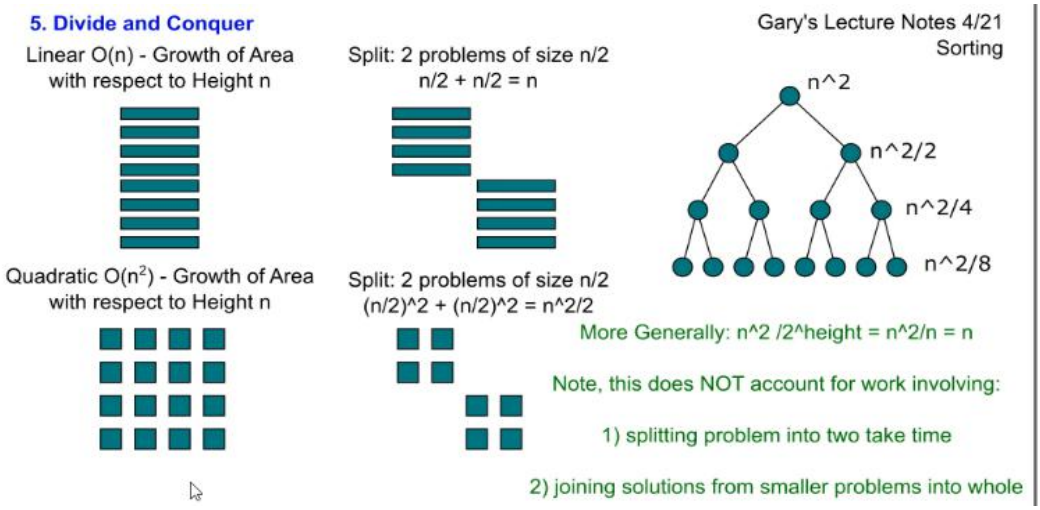
```java
void merge(int[] array, int left, int mid, int right) {
    // calculating lengths
    int lengthLeft = mid - left + 1;
    int lengthRight = right - mid;

    // creating temporary subarrays
    int leftArray[] = new int [lengthLeft];
    int rightArray[] = new int [lengthRight];

    // copying our sorted subarrays into temporaries
    for (int i = 0; i < lengthLeft; i++)
        leftArray[i] = array[left+i];
    for (int i = 0; i < lengthRight; i++)
        rightArray[i] = array[mid+i+1];

    // iterators containing current index of temp subarrays
    int leftIndex = 0;
    int rightIndex = 0;

    // copying from leftArray and rightArray back into array
    for (int i = left; i < right + 1; i++) {
        // if there are still uncopied elements in R and L, copy minimum of the two
        if (leftIndex < lengthLeft && rightIndex < lengthRight) {
            if (leftArray[leftIndex] < rightArray[rightIndex]) {
                array[i] = leftArray[leftIndex];
                leftIndex++;
            }
            else {
                array[i] = rightArray[rightIndex];
                rightIndex++;
            }
        }
        // if all the elements have been copied from rightArray, copy the rest of leftArray
        else if (leftIndex < lengthLeft) {
            array[i] = leftArray[leftIndex];
            leftIndex++;
        }
        // if all the elements have been copied from leftArray, copy the rest of rightArray
        else if (rightIndex < lengthRight) {
            array[i] = rightArray[rightIndex];
            rightIndex++;
```

```java
public static void mergeSort(int[] array, int left, int right) {
    if (right <= left) return;
    int mid = (left+right)/2;
    mergeSort(array, left, mid);
    mergeSort(array, mid+1, right);
    merge(array, left, mid, right);
}
```

9 4 2 3 8 5 1 7
4 9 2 是最先 copy 到 temporary array 的 values



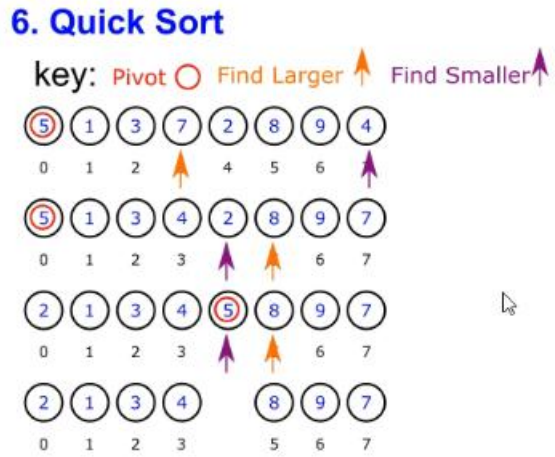
Merge sort 是分治法(Divide and Conquer)的一种经典应用



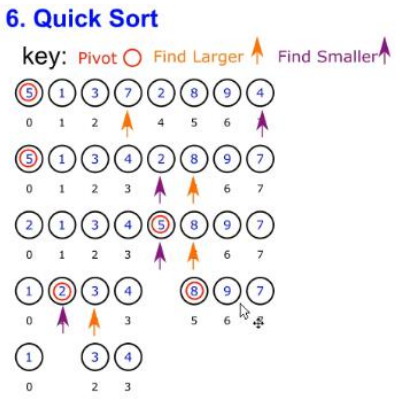
五．Quick sort

worst O(n^2)把最大或最小的 element 作为 pivot--inner loop O(n),recursive O(n)
所以 element 很多的时候用 quick 好，不容易是 O(n^2)
best/average O(n*log2n)　选 pivot 时候正好是中位数 median
unstable
也运用了 divide and conquer--分治法

先选择一个数作为 pivot 开始 partition，把比 pivot 小的 element 放 pivot 左边，大的放右边，再进行 recursive call 来排序

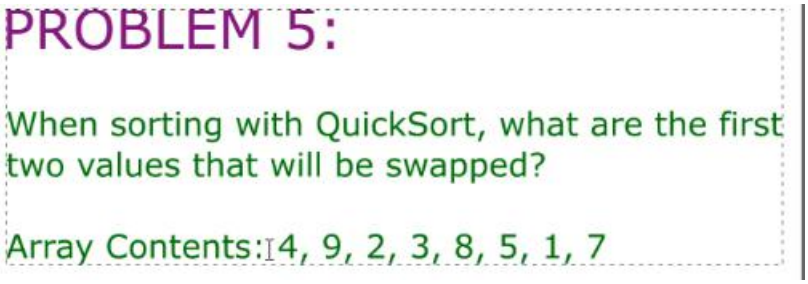


1. 任意选择一个数作为 pivot--5
2. 从左到右开始找比 5 大的数--7 设置成 Find Larger
3. 在 7 的右边从右往左找有没有比 5 小的数(如果没有则 partition 完成),把比 5 小的数设置成 Find Smaller
4. 把 Find Larger 和 Find Smaller 的数互换位置
5. 之后重复上面从左往右找比 5 大的第一个数作为 Find Larger--8，从 8 的右边从右往左找比 5 小的数--这次没有了--也就是 Find Smaller cross over Find Larger
6. partition 完成，准备开始 recursive
7. 在 recursive 之前再做一次交换，把 pivot 和 Find Smaller 的 element 互换
8. 开始 recursive call on 2 1 3 4，另一个 recursive call on 8 9 7--对 2134 设置 pivot 重复上面步骤




```
Algorithm: quickSort(array,start,end)
if(end-start < 2) swap when necessary
choose pivot = ??? (start)
// partition array
while (orange left of < purple)
   search left to right for orange larger than pivot
   search rigth to left for purple smaller than pivot
   if(orange left of< purple)
      swap contents of orange and purple
swap pivot with purple
quickSort(array,start,purple-1)
quickSort(array,purple+1,end)
```

base case: if(end-start<2)



4 是 pivot，9 和 1 是 first two values 被 swapped

```java
static int partition(int[] array, int begin, int end) {
    int pivot = end;

    int counter = begin;
    for (int i = begin; i < end; i++) {
        if (array[i] < array[pivot]) {
            int temp = array[counter];
            array[counter] = array[i];
            array[i] = temp;
            counter++;
        }
    }
    int temp = array[pivot];
    array[pivot] = array[counter];
    array[counter] = temp;

    return counter;
}

public static void quickSort(int[] array, int begin, int end) {
    if (end <= begin) return;
    int pivot = partition(array, begin, end);
    quickSort(array, begin, pivot-1);
    quickSort(array, pivot+1, end);
}
```

六．Bubble sort

best/worst O(N)--用 nested loop　stable

```
BubbleSort(numbers, numbersSize) {
   for (i = 0; i < numbersSize - 1; i++) {
      for (j = 0; j < numbersSize - i - 1; j++) {
         if (numbers[j] > numbers[j+1]) {
            temp = numbers[j]
            numbers[j] = numbers[j + 1]
            numbers[j + 1] = temp
         }
      }
   }
}
```

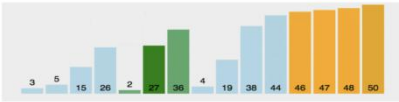
index012345，0 和 1 先比较，再 1 和 2 比较，2 和 3 比较，每次比较都会 sort，这样一对对比较到最后一个 index，最后一个 index 就是最大的 element(sorted part)
再从头开始开始一对对比较，每次比较都会放一个 unsorted part 里最大 element 到 sorted part

冒泡排序（Bubble Sort） 最为简单的一种排序，通过重复走完数组的所有元素，通过打擂台的方式两个两个比较，直到没有数可以交换的时候结束这个数，再到下个数，直到整个数组排好顺序。因一个个浮出所以叫冒泡排序。双重循环时间 O(n^2)

算法描述：

1. 比较相邻两个数据如果。第一个比第二个大，就交换两个数
2. 对每一个相邻的数做同样1的工作，这样从开始一队到结尾一队在最后的数就是最大的数。
3. 针对所有元素上面的操作，除了最后一个。
4. 重复1~3步骤，知道顺序完成。

代码可视化：



七. **Radix sort** 基数排序--为 Integer 设计

best/worst:O(k*n) k 是数字的最大位数　stable　out-place
把一个 array 里按 element 的数字的最大位数来放到一个 0-9 buckets 里
比如要 sort 1 2 34 53 56 88 91
1.
则放到 0-9 buckets 则是(按照个位数放)

	91								
	1	2	53	34	56			88	
0	1	2	3	4	5	6	7	8	9---这行为不同的 bucket

2.再放回去
91 1 2 53 34 56 88　--同一个 bucket 的数会先把底下的数放回去，这样保证 stable

3.再按照十位数放

					56				
	1	2	34		53			88	91
0	1	2	3	4	5	6	7	8	9---这行为不同的 bucket

4.再放回去
1　2　34　53　56　88　91　sort 完成

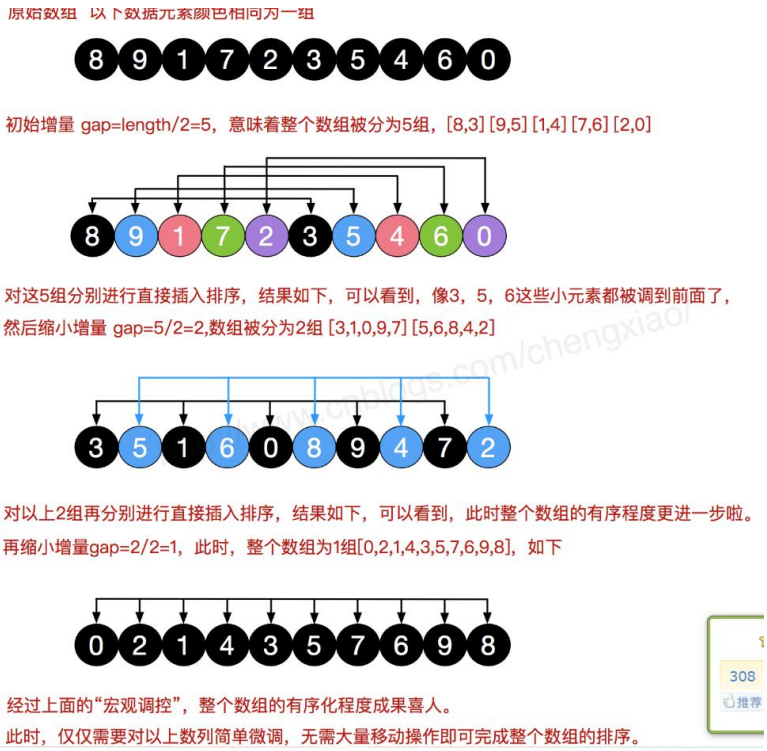
从 1’ digit 开始，再 10’s digit, 100’s digit.....
如果所有数都是两位数，但用 100’s digit 来排，则每个数都在 bucket 0

```java
/**
 * 基数排序
 * @param array
 * @return
 */
public static int[] RadixSort(int[] array) {
    if (array == null || array.length < 2)
        return array;
    // 1.先算出最大数的位数;
    int max = array[0];
    for (int i = 1; i < array.length; i++) {
        max = Math.max(max, array[i]);
    }
    int maxDigit = 0;
    while (max != 0) {
        max /= 10;
        maxDigit++;
    }
    int mod = 10, div = 1;
    ArrayList<ArrayList<Integer>> bucketList = new ArrayList<ArrayList<Integer>>();
    for (int i = 0; i < 10; i++)
        bucketList.add(new ArrayList<Integer>());
    for (int i = 0; i < maxDigit; i++, mod *= 10, div *= 10) {
        for (int j = 0; j < array.length; j++) {
            int num = (array[j] % mod) / div;
            bucketList.get(num).add(array[j]);
        }
        int index = 0;
        for (int j = 0; j < bucketList.size(); j++) {
            for (int k = 0; k < bucketList.get(j).size(); k++)
                array[index++] = bucketList.get(j).get(k);
            bucketList.get(j).clear();
        }
    }
    return array;
```

Sorting signed integers
array 里有 negative integer 则要 2 个 bucket 分别放 negative integer 和 non-negative integer，再把 negative integer 用 reverse order 和 non-negative integer 放到一起

八．Shell sort

best/worst O(nlog n)　unstable



先选一个 gap value，之后每次除以 2
array size=6
gap value=2，则有 2 个 interleaved list，每个 list 有 3 个 element
gap value=4，则是 2 个有 2 个 element 的 list 和 2 个只有 1 个 element 的 list

```
InsertionSortInterleaved(numbers, numbersSize, startIndex, gap) {
   i = 0
   j = 0
   temp = 0  // Temporary variable for swap

   for (i = startIndex + gap; i < numbersSize; i = i + gap) {
      j = i
      while (j - gap >= startIndex && numbers[j] < numbers[j - gap]) {
         temp = numbers[j]
         numbers[j] = numbers[j - gap]
         numbers[j - gap] = temp
         j = j - gap
      }
   }
}
```

```
ShellSort(numbers, numbersSize, gapValues) {
   for each (gapValue in gapValues) {
      for (i = 0; i < gapValue; i++) {
         InsertionSortInterleaved(numbers, numbersSize, i, gapValu
      }
   }
}
```

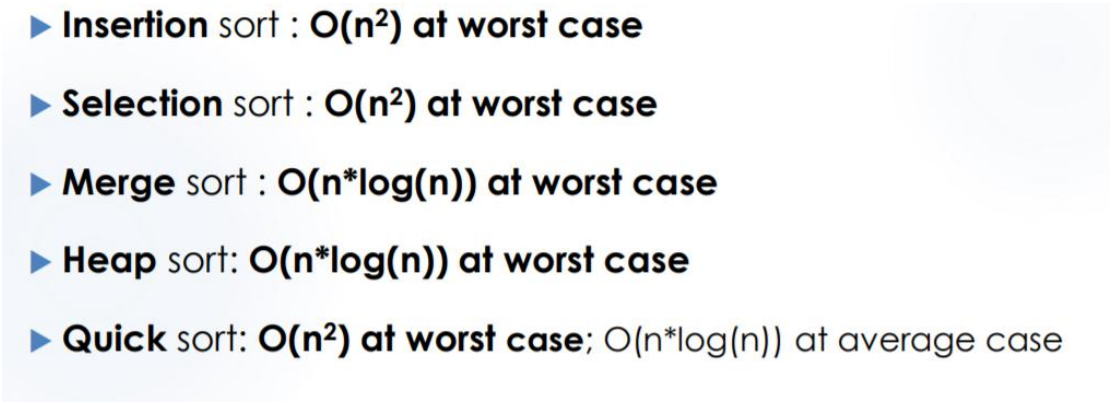
九．Bucket Sort

best：O(n+k) worst：O(n+k) Average：O(n^2) stable
每个 bucket 都有额外的 sorting algorithm
non-negative 数字的 array 才能用 bucket sort

bucket index = [number*(N/(M+1))]
number 是 array 里的 element，N 是 bucket 的数量，M=99？
list 里有(71,22,99,7,14)
其中　7*5/(99+1)=0
　　14*5/(99+1)=0
则 7 和 14 都放在 bucket 0--index0

```java
public static ArrayList<Integer> BucketSort(ArrayList<Integer> array, int bucketSize) {
    if (array == null || array.size() < 2)
        return array;
    int max = array.get(0), min = array.get(0);
    // 找到最大值最小值
    for (int i = 0; i < array.size(); i++) {
        if (array.get(i) > max)
            max = array.get(i);
        if (array.get(i) < min)
            min = array.get(i);
    }
    int bucketCount = (max - min) / bucketSize + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketCount);
    ArrayList<Integer> resultArr = new ArrayList<>();
    for (int i = 0; i < bucketCount; i++) {
        bucketArr.add(new ArrayList<Integer>());
    }
    for (int i = 0; i < array.size(); i++) {
        bucketArr.get((array.get(i) - min) / bucketSize).add(array.get(i));
    }
    for (int i = 0; i < bucketCount; i++) {
        if (bucketSize == 1) { // 如果带排序数组中有重复数字时  感谢 @见风任然是风 朋友指出错误
            for (int j = 0; j < bucketArr.get(i).size(); j++)
                resultArr.add(bucketArr.get(i).get(j));
        } else {
            if (bucketCount == 1)
                bucketSize--;
            ArrayList<Integer> temp = BucketSort(bucketArr.get(i), bucketSize);
            for (int j = 0; j < temp.size(); j++)
                resultArr.add(temp.get(j));
        }
    }
    return resultArr;
}
```

Comparison of sorting algorithm
1.Merge sort 比 Quick sort 需要 extra memory，因为每次 call merge 都会 create 新的 array 去放
2.selection sort 的 swap 次数最少
3.对于 mostly sorted array，insertion sort 最好



3.只有 merge sort 不是 in place，因为要 extra memory
4.只有 Insertion 和 merge sort 是 stable

Overview：
1.fast sorting algorithm 是 average runtime complexity 为 O(N*logN)

Table 3.10.1: Sorting algorithms' average runtime complexity.

Sorting algorithm	Average case runtime complexity	Fast?
Selection sort	O(N^2)	No
Insertion sort	O(N^2)	No
Shell sort	O($N^{1.5}$)	No
Quicksort	O($NlogN$)	Yes
Merge sort	O($NlogN$)	Yes
Heap sort	O($NlogN$)	Yes
Radix sort	O(N)	Yes

fast: Radix, Quick, Merge, Heap